Application Development in Java

From OOP to SOP

1. Foreword

During development of software in the past years, I have come across the same problems again and again:

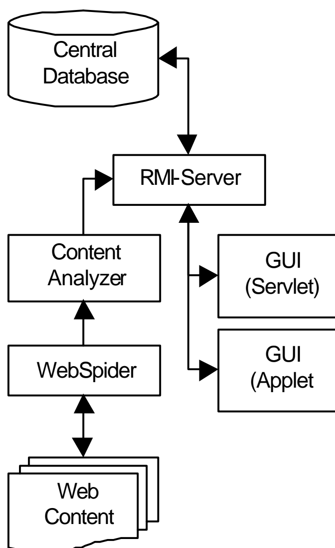
- Changing requirements during the implementation phase.
- Requests for additional features throughout all development phases.

Current OOD/OOP (in Java) is ideal for designing and implementing a fixed set of ideas. Enhancements beyond what was planned are also possible, as long as the enhancement remains within the scope of the model.
But when it comes to radically changing the specifications, adding concerns which are as important as those already in the model or the need to change the contents of different distributions, OOD/OOP gets into trouble.
When I learned about the multi-dimensional separation of concerns and Hyper/J, I was fascinated with the prospects.
While a careful design is still neccessary, it can be limited to smaller parts of the whole, making the design more flexible as a whole. It also enables the developers to create classes with a higher level of reusability.
In this paper, I'd like to give you some insight in one of my current projects which has been designed with SOP in mind.

Here, you'll fins a brief description of the core features of the application I want to use to show you what difference SOP makes to me instead of OOP.

2. Description:

All applications start small...



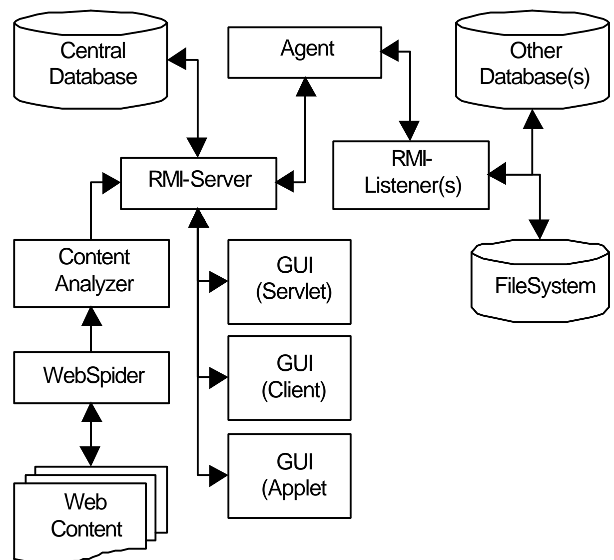
The application described here is a „smart“ proxy server for browsing the intra- and internet.
It's main task is information gathering and evaluation.
The second task it has to perform is to provide the user with a simple, yet powerful user interface for navigating the net.
The following capabilities were specified in the planning phase:

- Scan HTML files from the intra- and internet.
- Cache these files on a local RAID array in a customized directory structure, keeping those pages used more often or more intensively while discarding others.
- Analyze the structure of all pages scanned and extract links leading to other pages to automatically scan these as well.
- Analyze the content of the page and add all expresions to a searchable index.
- Generate a walkable, enhanced sitemap to allow easier navigation.
- Limit the number of pages visible to the user to those with the highest probability of success for finding the information desired. Optionally, show all pages directly connected to these pages.
- Keep a user history, search these pages first when checking for an information.

The changes to the system through implementation phase are described here.

3. Changes of the model during the Implementation Phase

...and expand until they can send mail



During implementation phase the following capabilities were added to our To-do list:

- Use of mobile agent technology to gain access to databases throughout a network as additional references for searching and browsing. This included the deploymant of RMI-Listeners on each server which should be included in the proxy-system.

- A three dimensional user interface for navigation.
- Addition of a centralized logging tool, to log all events from all RMI-Servers and –Listeners connected to the proxy system.
- Dynamic logging facilities.
- Enhancements to the way information is analyzed by the proxy-system through use of regular expressions.

These changes normally would have made it necessary to completely change the design of the software, adding additional abstract layers of classes to merge all capabilities into a reusable model.

4. List of Concerns

In the first phase of development, the following concerns were taken care of:

- Information Gathering
- Information Analysis
- Persistence
- User Interface
- Logging

The additional features amounted to the these new concerns:

- Advanced Information Analysis
- Mobile Objects
- Advanced Logging

While the first concern is merely an enhancement, the second and third part entail a great deal of redesign... or rather would entail that, were this the standard OOP way.
In the following chapter, you'll find information on how the application is structured now.

5. General Structure:

The server is divided into different packages desctibed here:

RMI-Server

- Gather all data from the NetAgent and ContentAnalyzer and store them in binary trees for quick location of information.

RMI-Listener

- Generate binary trees for information stored in specific directories not generally available to the WebSpider.
- Allow access to these trees and also to any database available on the system.
- Enable agents to gather information.

NetSpider

- Walk over a start page and from there follow every link up to a predetermined depth.
- Download all HTML and CSS files and store them in the file system.

NetAgent

- Retrieve data from throughout the available intranet-structure and send it to the RMI-Server for incorporation into the binary tree.

Parser

- Parse HTML pages gathered from the netspider and adjusting any links to fit the directory structure of the cache keeping the files.

ContentAnalyzer

- The content analyzer is based on a predetermined set of rules which lie in a resource bundle and are parsed into regular expressions for later use.
- Strip all HTML files of their tags and generate a red-black tree to store the location of any word existing in all HTML files.
- Removing all expressions existing in a dictionary of common words.

Persistence

- Persist the HTML files and graphics to disk.
- Persist the binary trees to disk or database.

UserInterface.data

- This package contains all classes necessary to generate a representation of all documents currently visible to the user (limited visibility to reduce memory comsumption and overhead).

UserInterface.view

- The user interface is based on three different modes:
  - a Servlet to send HTML pages to browsers,
  - an Applet to generate an AWT based view
  - a Swing-based client working with mobile objects to generate the GUI
- Generate a graphic view on the documents scanned, their relation among themselves and additional data (size, last changed, response time of not yet cached files, ect.)
- Generate different views, depending on what elements the user requested.

This structure is augmented by the following packages:

Logging

- Log all events with timing information for optimization and process analysis throughtout the network of servers and RMI-Listeners available to the system.

Timer

- Generate Timer objects containing specific information on how long which task takes

UtilityClasses

- As we currently use Java 1.1.8, many classes are less than optimal in performance.
  To alleviate that problem, we wrote optimized versions of many structures, like StringBuffers, Piped Streams, Vectors, Hashtables, ect.

Additional Note:
This application uses the mobile objects to encapsulate data (and user interface where necessary) and transmitting them together via RMI. Namely, the UserInterface and the NetAgent take advantage of that technology.

6. Changes to the wishlist and their implementation

This application gradually gained complexity through planning and development by adding the following aspects to the model:

1. **Variable logging, depending on the location the application runs.**
   1.1. **Problem:** On different machines, different logging output was required and different methods / classes were under observation
   1.2. **OO-Solution:** This could have been achieved through insertion of logging / debugging code into all methods we wanted to take a look at, but in general this is not easy to read, re-use or maintain.
   1.3. **SOP-Solution:** Basically, we worked along the lines of the example in the Hyper/J manual.

2. **Logging of different areas of interest to different logfiles on separate machines**
   2.1. **Problem:** As the application grew, logging became more complex and complicated. Different logfiles were required, sometimes, information was to be stored in several logfiles at once.
   2.2. **OO-Solution:** Insertion of even more logging code in every relevant method. Furthermore, varying concerns would require changing the logging / debug statements pretty often.
   2.3. **SOP-Solution:** We just enhanced the classes in the logging hyperslice and bracketed the classes / methods to be logged with several logging calls.

3. **Logging of many different areas of interest to a centralized set of logfiles on one central machine**
   3.1. **Problem:** The addition of the RMI-Listeners and the broadening of their abilities added a further level of complexity to the whole theme as new classes had to be written to allow logging over the intranet and integration of all logging events into a coordinated form.
   3.2. **OO-Solution:** Generate new, network-aware and -enabled classes which subclass the current logging classes.
   3.3. **SOP-Solution:** We added the additional capability by merging the agent-concern with the logging concern, thus creating an agent able to transport logging information to a desired destination.

4. **Addition of RMI-Listeners**
   4.1. **Problem:** The RMI-Listeners were incorporated to allow agents to work with the databases available to the system. On each system available to the proxy server, a VM runs, listening to an RMI port, awaiting the arrival of an agent. The Listener offers a standardized interface to read data from all resources available to the Listener on the machine.
   This includes querying databases via SQL and scanning portions of the file system for data and reading from a red-black binary tree containing information generated by the ContentAnalyzer.
   4.2. **OO-Solution:** Basically, we'd have to write a new subclass, overriding most of the methods of the RMI-Server to generate the new class (or make the RMI-Listener the superclass, leaving most methods to be overwritten by the RMI-Server code).
   Another approach would be the inclusion of property-driven switches, allowing the VM to determine whether the application is a RMI-Listener or the RMI-Server.
   Both ways are possible and not too difficult to use.
   4.3. **SOP-Solution:** Generate a central class and add the additional functionality required for an RMI-Server as different concern.
   Thus, the codebase remains more clearly separated and easy to maintain.

5. **ContentAnalyzer using regular expressions**
   5.1. **Problem:** In the beginning of development, the rules for analyzing the content of the HTML files were hard-coded and limited to some easy to implement rules.
   As XML became more important, we needed to add some XML support as well. That didn't work well without regular expressions. As we came across a library which did just that, we chose to incorporate a regular-expression checking.
   This was to be performed on the RMI-server only as the library was quite slow and we had only one really well performing machine.
   5.2. **OO-Solution:** This normally would entail adding new classes and statements to existing classes to the package or generating a different package, implementing a different check algorithm in addition to the one already used.
   This is pretty impractical as the different ways of analyzing the documents require a different approach to the coding.
   5.3. **SOP-Solution:** Adding the check-classes for a normal checking to those of a reg-ex concern dding the check mechanism in the SOP way makes it easy to merge one

mechanism with another on demand for different systems.
This is necessary as the regular expressions library is pretty slow and should only be used in offline mode (i.e. on the RMI-Server).
Thus, merging the additional check packages into the already existing set by name, we'd be able to check with regular expressions on the RMI-Server while the RMI-Listeners use the faster, less exact scanning method.

6. **Different distributions of the same classes (RMI-Listener and RMI-Server)**

   Problem: As soon as we began implementing the RMI-Listeners, it became clear that we would need to double most methods from the RMI-Server, changing what we needed and adding what only the RMI-Listener needed.
   OO-Solution: Write new classes and subclass the RMI-Listener to use these new classes. This results in an increase in maintenance work as the work on the RMI-Server evolves.
   SOP-Solution: Add another hyperslice, of course. ☺

7. **3D User Interface**
   7.1. **Problem:** The user interface has to be changed from a conventional two-dimensional tree view to a three-dimensional, OpenGL based one. This was necessary as the structure of the links connecting the pages could not be displayed in a variable granularity in 2D.
   Thus, the 3D view had to be incorporated in addition to the other views.
   Normally, this would require extensive changes to the conventional tree model implemented.
   7.2. **OO-Solution:** Basically, this idea requires a redesign of all classes in the UserInterface.data to allow for a three dimensional web instead of a two-dimensional tree view.
   7.3. **SOP-Solution:** See 6.3

8. **Mobile Workspace**
   8.1. **Problem:** The ability to log on to the system from a different RMI-Listener. In that case, the last location of the User is to send the complete GUI to the new location, rebuilding it there completely, freeing up resources on the old location.
   8.2. **OO-Solution:** Generate a net-aware subclass of the GUI which is able to follow the user from workstation to workstation.
   8.3. **SOP-Solution:** Another concern, of course. Merge the Agent concern with the GUI and you have a GUI following the user.
   Of course, it wasn't that easy... A careful redesign of the Agent hyperslice allowed us to do just that.

Author: Juri Memmert
e-mail: memmert@ibm.net

Phone: +49 (0)172 / 9074037

Currently, I have a contract which keeps me from my office most of the month, thus, any mail sent to me should go to:

Juri Memmert
c/o Hotel Neue Stuben
Room #40
Bahnhofstraße 13
D-38442 Wolfsburg-Fallersleben